

Finding long simple paths in a weighted digraph using pseudo-topological orderings

Miguel Raggi *

Escuela Nacional de Estudios Superiores
Universidad Nacional Autónoma de México

September 26, 2016

Abstract

Given a weighted digraph D , finding the longest simple path is well known to be NP-hard. Furthermore, even giving an approximation algorithm is known to be NP-hard. In this paper we describe an efficient heuristic algorithm for finding long simple paths, using an hybrid approach of DFS and pseudo-topological orders, a generalization of topological orders to non acyclic graphs, via a process we call “opening edges”. An implementation of this algorithm won the Oracle MDC 2015 coding competition.

1 Introduction

We focus on the following problem: Given a weighted digraph $D = (V, E)$ with $w : E \rightarrow \mathbb{R}^+$, find a simple path with high weight. The **weight** of a path is the sum of the individual weights of the edges belonging to the path. A path is said to be **simple** if it contains no repeated vertices.

The problem of finding long paths in graphs is well known to be NP-hard, as it is trivially a generalization of HAMILTON PATH. Furthermore, it was proved by Björklund, Husfeldt and Khanna in [BHK04] that the longest path cannot be approximated in polynomial time within $n^{1-\epsilon}$ for any $\epsilon > 0$ unless $P = NP$.

While LONGEST SIMPLE PATH has been studied extensively for simple graphs (for example in [Scu03], [ZL07], [PD12]), not many efficient algorithms exist for weighted directed graphs. A nice survey from 1999 can be found at [Sch99]. A more recent comparison of 4 distinct genetic algorithms for approximating the solution in weighted digraphs can be found in [PAR10].

*Research supported in part by PAPIIT IA106316, UNAM.

Our contribution lies in a novel method of improving existing paths, and an efficient implementation of said method. The proposed algorithm consists of two parts: finding good candidate paths using heuristic DFS and then improving upon those candidates by attempting to either replace some nodes in the path by longer subpaths—or simply insert some subpaths when possible—by using pseudo-topological orders.

As mentioned in the abstract, the implementation won the Oracle MDC coding competition in 2015. In the problem proposed by Oracle in the challenge “Longest overlapping movie names”, one needed to find the largest concatenation of overlapping strings following certain rules.

The full C++ source code can be downloaded from

<http://github.com/mraggi/LongestSimplePath>.

In section 2 we give some basic definitions. We describe the proposed algorithm in section 3. Finally, we give some implementation details and show the result of some experimental data in section 4.

2 Preliminaries

Definition 2.1. A **directed acyclic graph** (or DAG) D is a directed graph with no (directed) cycles.

In a directed acyclic graph, one can define a partial order \prec of the nodes, in which we say $v \prec u$ iff there is a directed path from v to u .

Definition 2.2. A **topological ordering** for a directed acyclic graph D is a total order of the nodes of D that is consistent with the partial order described above. In other words, it is an ordering of the nodes such that there are no edges of D which go from a “high” node to a “low” node.

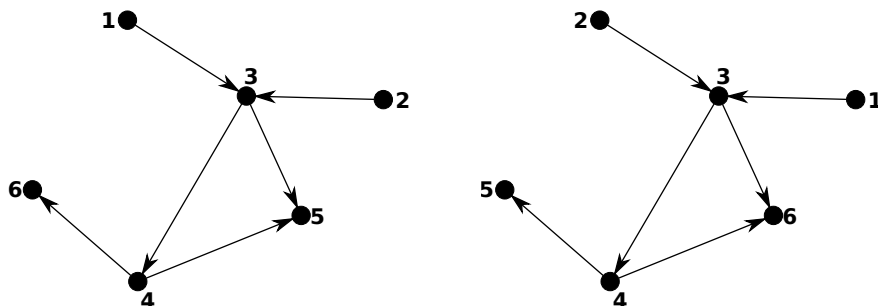


Figure 1: Two different topological orders of the same digraph

Definition 2.3. Given a digraph D , a **strongly connected component** C is a maximal set of nodes with the following property: For each pair of nodes $x, y \in C$, there exists a (directed) path from x to y and one from y to x . A **weakly connected component** is a connected component of the associated simple graph.

Definition 2.4. Given a digraph D , the **skeleton** S of D is the graph constructed as follows: The nodes of S are the strongly connected components of D . Given $x, y \in V(S)$, there is an edge $x \rightarrow y$ iff there exists $a \in x$ and $b \in y$ such that $a \rightarrow b$ is an edge of D .

One can easily observe that S is a directed acyclic graph.

Denote by \bar{v} the connected component of D which contains v . Given a node v , we define the **out-rank** of v as the length of the longest path of S that *starts* at \bar{v} . Similarly, we define the **in-rank** of v as the length of the longest path of S which *ends* at \bar{v} .

2.1 Longest simple path on DAGs

In case that the digraph is a acyclic, a well-known algorithm that uses dynamic programming can find the optimal path in $O(n)$ time. The idea is that if there are no cycles, simply taking the negative of the weights and using any of the algorithms for finding the shortest path works.

As this algorithm is an essential building block of the algorithm described in section 3.2, we add a short description here for convenience. For a longer discussion see, for example, [SW11].

Given D a DAG, do:

1. Associate to each vertex v a real number x_v , (which will end up representing the longest simple path that *ends* at v).
2. Find a topological ordering of the nodes of D .
3. In said topological order, iteratively set x_v to the maximum value of $x_p + w(p \rightarrow v)$, where p runs over all nodes which have an edge to v and $w(p \rightarrow v)$ is the weight of the edge from p to v .
4. Once we have x_v for every node v , reconstruct the path by backtracking from the node v with the highest x_v .

This algorithm is simple to implement and efficient. Its running time is $O(E + V)$, where E is the number of edges of D .

3 The Algorithm

In what follows we shall assume we have preprocessed the graph and found the weakly connected components, the strongly connected components, and have fast access to both

the outgoing edges and incoming edges for each node. Furthermore, we may perform the following algorithm on each weakly connected component, so without loss of generality, assume D is weakly connected.

Our proposed algorithm has two main parts: In the first part we find long paths using heuristic depth first search, choosing in a smart way which nodes to explore first, and in the second part we discuss a strategy to improve the paths obtained in the first part. The second part is where the main contribution of this paper lies.

3.1 Depth-first search

The first part is a variation of the standard depth-first search technique.

We explore the digraph in a depth-first manner, at each step making sure the explored paths do not repeat any nodes. We also do a forward search and a backward search (using incoming edges instead of outgoing edges).

3.1.1 Choosing the next node

So we are left with the following questions: At which node do we start our search at? And then, while performing DFS, which nodes do we explore first?

The first question is easily answered: start at nodes with high out-rank when performing forward search first and start at nodes with high in-rank when performing backward search.

We answer the second question first using a variety of information we collect on each node before starting the search:

1. The out-rank and in-rank.
2. The out-degree and in-degree.
3. A *score* described below.

Assume we are going to perform forward search. For backward search, do the same on the transpose digraph.

Once we find the score of each node, order them by rank first and score second, with some exceptions we will mention below. Order the out-neighbors of each node using the same order.

Formally, let k be a constant (small) positive integer. For each node v , let $A_k(v)$ be the sum of weights of all paths of length k starting at v . For example, $A_1(v)$ is the weighted out-degree.

Given a choice of parameters $a_1, a_2, \dots, a_k \in \mathbb{R}$, construct the (out) score for node v as

$$\text{score}_{\text{out}}(v) = \sum_{i=1}^k a_i A_i(v)$$

For a longer discussion on parameters a_i , see 4.

When performing forward search, perhaps counter-intuitively, giving high priority to nodes with low score (as long as it is not 0) consistently finds better paths than giving high priority to nodes with high score. The only explanation (perhaps not fully satisfying) we could muster is that exploring nodes with low score first means *saving* the good nodes—those with high scores—for later use, once more restrictions are in place.

In addition, we also use the in-degree information in an analogous way.

3.2 Pseudo-topological order

The idea behind the second part of the algorithm is to try to improve paths by either inserting some short paths in-between or replacing some nodes by some short paths in an efficient way that covers both.

We begin by introducing some definitions.

Definition 3.1. Given a digraph D , a **weak pseudo-topological ordering** \prec of the nodes of D is a total order of the nodes in which whenever $x \prec y$ and there is an edge $y \rightarrow x$, then x and y are in the same strongly connected component.

In other words, a weak pseudo-topological order is an order that is consistent with the partial order given by the skeleton *whenever it can be*.

Definition 3.2. Given a digraph D , a **strong pseudo-topological ordering** \prec of the nodes of D is a total order of the nodes in which whenever $x \prec y$ and there is an edge $y \rightarrow x$, every node in the interval $[x, y]$ is in the same strongly connected component.

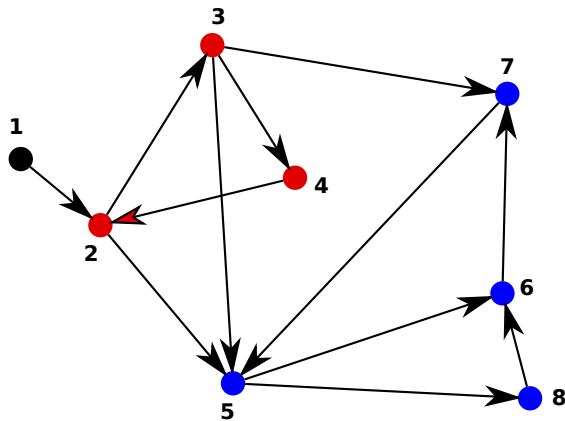


Figure 2: A (strong) pseudo-topological ordering

An easy way to get a random pseudo-topological ordering is to get a random topological ordering of the skeleton of the graph, and then “explode” the strongly connected components, choosing a random permutation of nodes in each component.

We can think of a pseudo-topological ordering as a topological ordering of a digraph in which we *erase* all edges that go from a “high” node to a “low” node. Therefore, we may apply the algorithm described in 2.1 to this new digraph, since it is acyclic.

However, the results obtained in this fashion are very poor compared to even random-order recursive depth-first search. See section 5.

It is only when combining the two approaches in a specific way that we get better paths.

3.2.1 Combining the two approaches

Impose the ordering of a good path on a pseudo-topological ordering. If we take a random pseudo-topological order T and a path P and permute the nodes of T so that those which are nodes of P appear in the same order as they appear in P , then we ensure that when running algorithm 2.1 we get a path that is at least as good as path P .

For example, say we start with path $P = 3 \rightarrow 1 \rightarrow 5 \rightarrow 8$. Construct a *random* pseudo-topological ordering, say, $T = (1, 8, 7, 4, 3, 6, 5, 2)$. Then impose the order defined by P into T , giving rise to $T' = (3, 1, 7, 4, 5, 6, 8, 2)$.

Order T' is obviously also a pseudo-topological order, since we know there is an edge from 3 to 1, which by the definition of pseudo-topological order, means 3 and 1 are in the same connected component, since 1 appears before 3 in T . The same can be said for each edge of P .

If after applying this technique we do find an improved path P' , we can repeat the process with P' , again taking a random pseudo-topological ordering, imposing the order of P' on this new ordering, and so on, until there is no more improvement.

This approach does indeed produce moderately better paths than pure DFS, even when starting from scratch, albeit taking longer. However, we can do (much) better.

3.2.2 Opening the edges

Again, we are in the setting where we have a path P which we wish to improve.

Now, instead of just imposing path P on multiple random pseudo-topological orders, construct orders as follows: Pick an edge $p_i \rightarrow p_{i+1}$ of path P and construct a random pseudo-topological order that is consistent with P and furthermore, for which p_i and p_{i+1} are as far apart as possible.

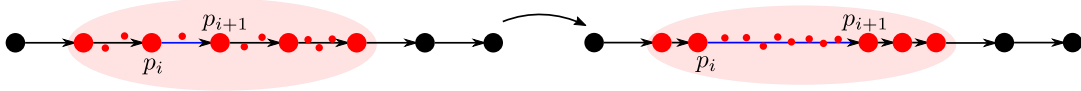


Figure 3: The process of opening an edge.

This is achieved by putting all nodes in the same connected component as the two nodes that make up the edge between p_i and p_{i+1} . In the figure above, the “large” nodes are nodes in P and the “small” nodes are all other nodes not in P in the same connected component as

p_i and p_{i+1} . If p_i and p_{i+1} are not in the same connected component, then, place *every* node in either connected component, and also every node that belongs in a connected component between the component of p_i and the component of p_{i+1} between the two nodes, in such a way that the order is still a pseudo-topological order.

We may repeat this process for each edge in P .

The process of opening an edge is relatively expensive, since we must run algorithm 2.1 each time.

We now make an attempt at explaining why opening edges works as well as it does. Consider:

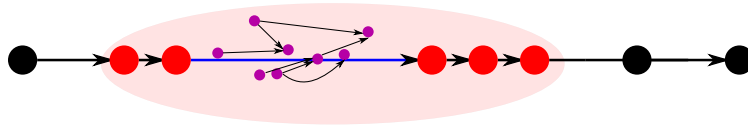
1. If there exist a node v that can be inserted into P , opening the corresponding edge surely finds it.
2. If there exists a node v that can replace a node p in P and make the path longer (by means of edge weights), this process will find it when opening the edge to either side of p .
3. Any (small) path that can be inserted into P will also be found if the corresponding nodes in the path happen to be in the correct order. This naturally leads us to the next section.

3.2.3 Opening the edges eXtreme

In the previous section, when opening up each edge, we put all the remaining nodes in the same connected component in a random order. We now consider the question of which order to give to those unused nodes. We discuss three different approaches: one heuristic that is quick and practical, one powerful and somewhat slow, and one purely theoretical, which uses sequence covering arrays.

Let B be the inbetween nodes. Since every other node will remain in their place, we face the problem of giving the inbetween nodes an ordering which maximizes the chances of finding a large improvement.

Consider the induced subdigraph on B :



Recall that running algorithm 2.1 with a pseudo-topological order is equivalent to finding the longest path on the digraph that results by erasing all the edges that go “backward”. Therefore, we must consider *only* orderings of B that are themselves pseudo-topological orders of the induced subgraph on B .

We describe three approaches to choosing an order of B .

3.2.4 The powerful approach

The “powerful” approach is to recursively repeat the process on the induced subgraph on B . That is, repeat for B the whole process of finding the strongly connected components, performing DFS as described in a previous section, finding suitable pseudo-topological orders, opening their respective edges, and recursively doing the same for the induced subgraphs in order to find good pseudo-topological orders and so on.

The problem with this approach is that with most standard graph data structures, we would need to reconstruct the induced digraph (*i.e.* rename the vertices of B but keep track of their original names), and the whole process is slow. Of course, we would only need to do this process once per connected component and then we can use the results for each edge of the path. See section 4.

3.2.5 The heuristic approach

Instead of attempting to repeat the whole algorithm on the induced subgraph, we try to mimic heuristically its results. Consider the following operation on the inbetween nodes: pick a node u at random, and exchange its position with the first out-neighbor v of u which appeared before u in the pseudo-topological order. If no such neighbor exists, simply find pick another u .

Repeat this operation (which is quite inexpensive) a few thousand times. The following theorem ensures that this process will likely end in a (weak) pseudo-topological order.

Theorem 3.3. With probability approaching 1 as the number of times approaches infinity, the order constructed above is a weak pseudo-topological order of B .

Proof: For any digraph, given a total order of the nodes, call a pair of nodes (a, b) *bad* if a appears before b in the order, there is a path from b to a but not one from a to b . In other words, if the strongly connected component which contains b is (strictly) less than the strongly connected component which contains a in the partial order of the skeleton of D .

Thus, we only need to prove that the number of bad pairs never increases after an operation, and that with some positive probability, it decreases.

Suppose we do an operation, and u is the chosen node, which is exchanged with v (so u was after v in the order, but there is an edge from u to v). Then only pairs of nodes of the form (a, u) and (v, a) could have changed status (and only when a is between u and v in the order).

Let U, V, A be the strongly connected components containing u, v, a respectively. If A is before U , then indeed the pair (a, u) is now bad after the exchange, but since we are assuming there is an edge from u to v , if A is before U , then it is before V , and so the pair (v, a) used to be bad, but is now good. When (v, a) becomes bad, the process is analogous, and (a, u) becomes good. ■

The above process then gives a random approximate algorithm to calculate weak pseudo-topological orders without calculating the strongly connected components.

3.2.6 The theoretical approach

Suppose we wanted to ensure that the found path P was locally optimal in the sense that no path with k nodes (for a sensibly small value of k) could be inserted into any part of P (perhaps replacing some subpath of P) to make it longer.

This problem can be restated as follows: we wish for a minimal set of permutations of S_n for which every k -subset of n appears in every possible order. This problem has been worked on by Colbourn *et al.* on [CCHZ13] or [MC14], where they give a mostly impractical algorithm for finding covering arrays, that works in practice up to $n \approx 100$. In the same paper, however, an easy probabilistic argument states that taking $O(\log(n))$ permutations randomly gives a non-zero chance of ending up with a covering array. This suggests that merely taking many random permutations would yield (probabilistically) the desired result. Unsurprisingly, this approach is not nearly as efficient as the other two.

For $k = 2$, however, a covering array is easy to find: take any permutation and its reverse.

4 Implementation details

4.1 Preprocessing the graph

The data structure we use for storing the graph is a simple pair adjacency list: one for the out-neighbors and one for the in-neighbors, so we can access either efficiently. The nodes are numbered from 0 to $n - 1$ and we store an array of size n , each element of which is an array storing the neighbors of the corresponding vertex. We store the weights in a matrix, for fast access. Since we will not be modifying the graph constantly, we found this to be most efficient speedwise.

Next, we find connected components. While it is true that finding the weakly connected components, strongly connected components and skeleton might require some non-trivial processing, this pales in comparison to the NP-hard problem of finding the longest path, and an efficient implementation (for example, using C++ boost graph library) can find the components on graphs with 30,000 nodes in less than a second.

Next, we find the out-heuristic and the in-heuristic scores for each node, as described in section 3.1, and sort the neighbors of each node according to the heuristic.

In our experiments, the whole preprocessing step, took about 0.2 seconds on the Oracle graph, which has $\sim 13,300$ vertices. This time includes reading the file of strings, figuring out which concatenations are possible and constructing the graph. Experiments with randomly generated graphs take less than 0.1 seconds if the graph is already in memory.

If one has a training set of a class of graphs, one could use some rudimentary machine learning to try to find the optimal parameters so that on average good paths are found. In fact, for the contest, we did just that, which provided a slight boost. The code includes a trainer for this purpose, but experimental results on the benefits of this are sketchy at best and do not (usually) warrant the long time it takes to train.

4.2 Efficient DFS

A straightforward implementation of depth-first search will not perform as well as it could in this case. The classical stack-based implementation (using a data structure sometimes denominated “frontier” or “fringe”) and its recursive cousin (which uses the call-stack as the frontier) are not very viable in this case, since one needs to keep track of the full path, and the memory requirements for storing this extra information make the algorithm run somewhat slow.

There is, however, a faster implementation in practice which requires only a little more information on the structure of the graph, namely that the neighbors of each node have a specified (and knowable) order beforehand, which happens almost always (and certainly for this problem).

We give here a quick description of a fast implementation of DFS. The knowledgeable reader may choose to skip the rest of this section.

Construct a function called “NextPath” which takes as input a (non-constant reference to) a path and modifies it *in place* to get the next path that would be explored when doing standard depth-first search. This function should return a boolean value denoting whether or not this is the very last path in the search, so one knows when to stop. This “next path” is usually a path in which one adds a single new node to the current path, but often we must backtrack until we can proceed with a new node.

In order to do so efficiently, we require a preset ordering of the out-neighbors of a given node, in order to select the next non-explored node.

The main reason this implementation is faster is that it does not require any extra memory allocations, aside from the memory required to store the longest possible path.

There are a few practical considerations:

- Keep track of the used nodes in a path so as to avoid repeated nodes. We found experimentally that a simple array of booleans the size of the number of nodes in the graph is most efficient for this purpose. Otherwise, use a set.
- Always save the best path found to date.
- Pick a node at random and perform forward depth first search, then erase the first few nodes and perform backward depth-first search (*i.e.* using the transpose digraph)
- Stop when no improvement has been made for a specified amount of time (usually a few dozen milliseconds).

Experimentally, we found that it doesn’t matter much which node one starts at, as long as one takes the considerations mentioned above (*i.e.* erasing the first few after forward search and then doing backward search, or vice-versa). For more robustness, start at a few different nodes, since each search is relatively cheap.

4.3 Pseudo-Topological orders

Once we have a pseudo-topological order T , we construct its inverse for fast access, so in addition of being able to answer the query “which vertex is in position i of T ?” we can also answer the query “at which position is vertex v in T ?” efficiently. Therefore, any operation we do on T must be mirrored to the inverse.

In addition, since we are constantly changing T and having to rerun algorithm 2.1, it is worth it to store x_v for each v , and just reprocess from the first vertex whose order was modified and onwards.

Fortunately, since much of the order has been preserved, we can use this information and just recalculate from the first modification onwards, speeding up the calculation considerably.

Finally, opening the edges is just a matter of rearranging the nodes to satisfy the condition which is straightforward. We found experimentally the process is much quicker if the edges of the path are opened in a random order and not sequentially.

Our implementation of the “powerful” approach described in 3.2.4 was by constructing a completely new graph and running the algorithm on the subgraph recursively, and so it was prohibitely slow, although it did tend to find longer subpath insertions with fewer edge openings. Perhaps this can be improved. The implementation of the heuristic approach of 3.2.5 was considerably more efficient over the random approach described in 3.2.6.

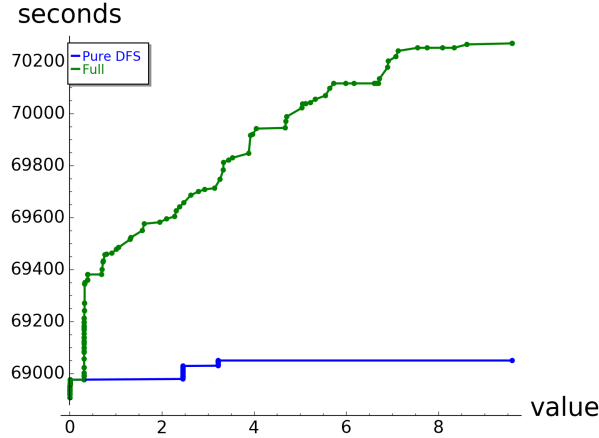
5 Experimental data

5.1 Self tests

We begin by benchmarking some of the different variations of the algorithm described in this paper. All tests below were conducted with an quad core Intel Core i7-4820K processor running on a single core with 8Gb DD3 RAM at 1600Mhz, with GCC 5.2 on Linux.

All graphs represent the following: the horizontal axis represents time (measured in seconds) and the vertical axis represents the total weight of the best path found after t seconds. While theoretically the functions should be non-decreasing, the slight variation is due to taking averages of multiple running times.

First, here is a test of pure DFS versus doing the full pseudo-topological order dance on a random Erdős-Renyi digraph with 10,000 vertices with $p = 0.0001$.



6 Acknowledgements

The author would like to thank the organizers of the Oracle MDC coding challenge for providing a very interesting problem to work on (and for the prize of a drone and camera, of course). Furthermore, the author would like to thank the other participants, specially Miguel Ángel Sánchez Pérez and David Felipe Castillo Velázquez for the fierce competition. Also, Marisol Flores for their comments on the paper. Lastly, this research was partially supported by PAPIIT IA106316.

References

- [BHK04] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna, *Approximating longest directed paths and cycles*, Automata, Languages and Programming, Springer, 2004, pp. 222–233.
- [CCHZ13] Yeow Meng Chee, Charles J Colbourn, Daniel Horsley, and Junling Zhou, *Sequence covering arrays*, SIAM Journal on Discrete Mathematics **27** (2013), no. 4, 1844–1861.
- [MC14] Patrick C Murray and Charles J Colbourn, *Sequence covering arrays and linear extensions*, Combinatorial Algorithms, Springer, 2014, pp. 274–285.
- [PAR10] David Portugal, Carlos Henggeler Antunes, and Rui Rocha, *A study of genetic algorithms for approximating the longest path in generic graphs*, Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on, IEEE, 2010, pp. 2539–2544.
- [PD12] Quang Dung Pham and Yves Deville, *Integration of ai and or techniques in constraint programming for combinatorial optimization problems: 9th international*

- conference, cpaor 2012, nantes, france, may 28 – june1, 2012. proceedings*,
 ch. Solving the Longest Simple Path Problem with Constraint-Based Techniques,
 pp. 292–306, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [Sch99] John Kenneth Scholvin, *Approximating the longest path problem with heuristics: a survey*, Ph.D. thesis, University of Illinois at Chicago, 1999.
- [Scu03] Maria Grazia Scutella, *An approximation algorithm for computing longest paths*,
 European Journal of Operational Research **148** (2003), no. 3, 584–590.
- [SW11] R. Sedgewick and K. Wayne, *Algorithms*, Pearson Education, 2011.
- [ZL07] Zhao Zhang and Hao Li, *Algorithms for long paths in graphs*, Theoretical Computer Science **377** (2007), no. 1, 25–34.